Chapter 4
Threads & Concurrency

Da-Wei Chang

CSIE.NCKU

Source: Abraham Silberschatz, Peter B. Galvin, and Greg Gagne, "Operating System Concepts", 10th Edition, Wiley.

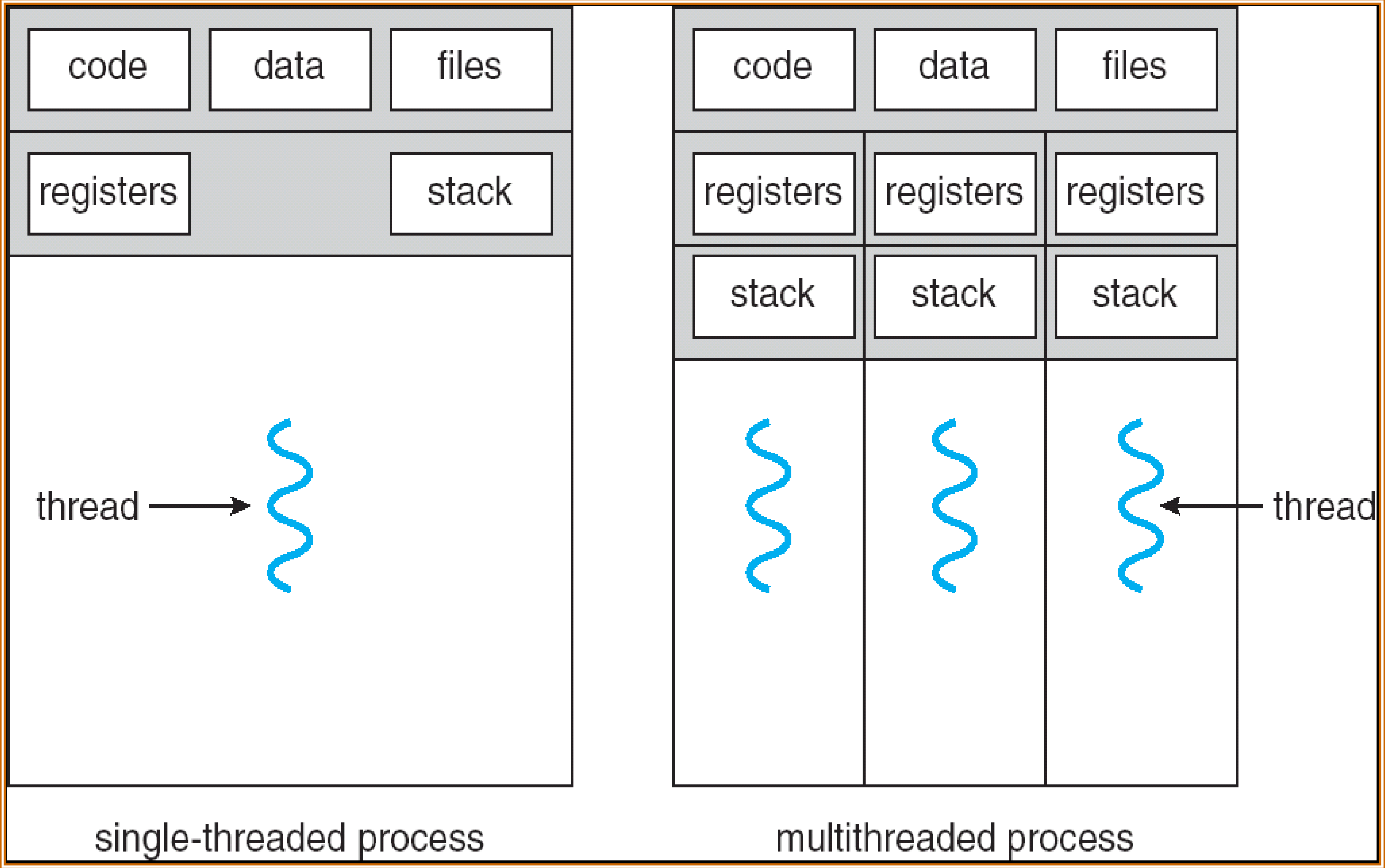
Outline

- Overview
- Multithreading Models
- Thread Libraries
- Threading Issues
- Operating-System Examples

Overview

- Most operating systems support multi-threaded processes
- A thread
  - A basic unit of CPU utilization
  - Comprises
    - Thread id
    - Program counter
    - Register set
    - Stack
- A multi-threaded process can perform more than one task at a time

Single and Multithreaded Processes



Motivation

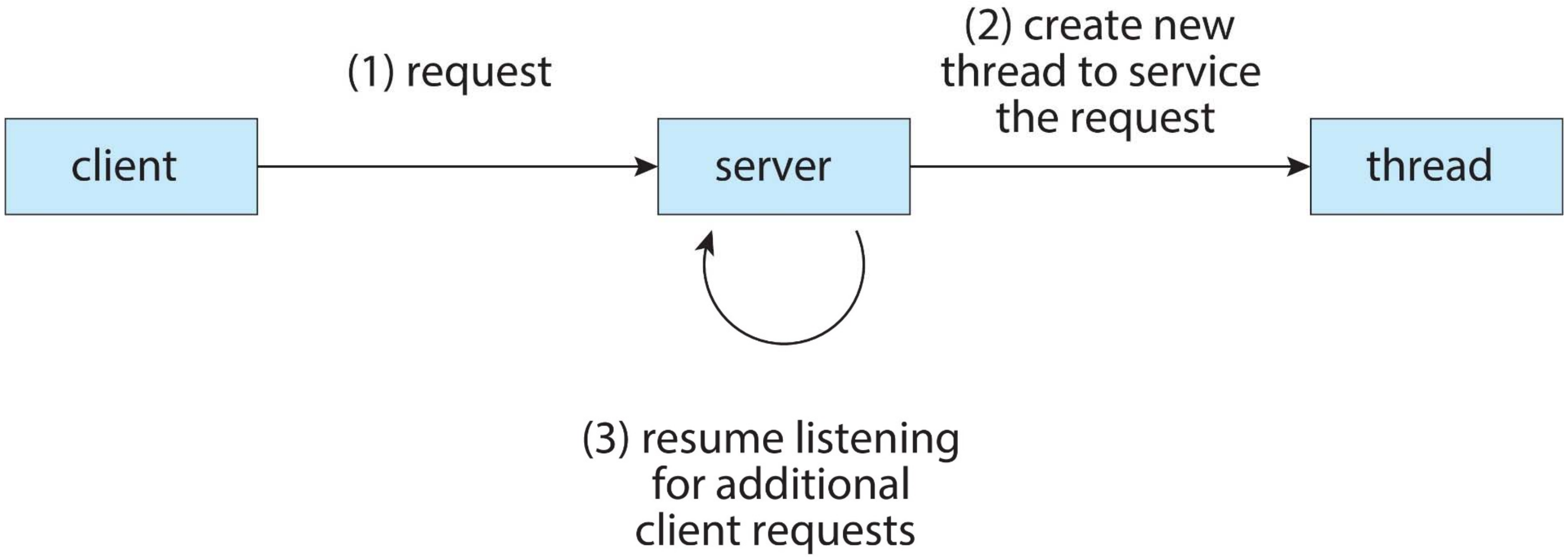
- A lot of software packages are multi-threaded
  - Web browser
    - One thread displays images/text
    - Another retrieves data from the network
  - Word processor
    - Display graphics
    - Respond to keystrokes
    - Perform spelling and grammar checking in the background
  - Web server
    - May have one thread for each request

Motivation

- If a web server is single-threaded
  - Serve only one client at a time
  - If it processes the requests one-by-one➔ a long waiting time
- Multi-process solution
  - Common before threads become popular
  - Process creation is time consuming and resource intensive
- Multi-threaded solution
  - More efficient
    - Threads are more lightweight than processes
- A multi-threaded web server may
  - Use a thread for listening for client requests
  - Create a thread for handling each request

    ** see the next slide*

Multithreaded Server Architecture



Motivation

- Threads are also important in RPC systems
  - RPC servers are multi-threaded
    - Serve each message via a separated thread
  - Java's RMI systems are also multi-threaded
- Many operating systems are multi-threaded
  - Each thread performs a specific task
    - E.g., Solaris has a set of threads for interrupt handling
    - E.g., Linux uses kernel thread(s) for writing back memory data to disk

Benefits

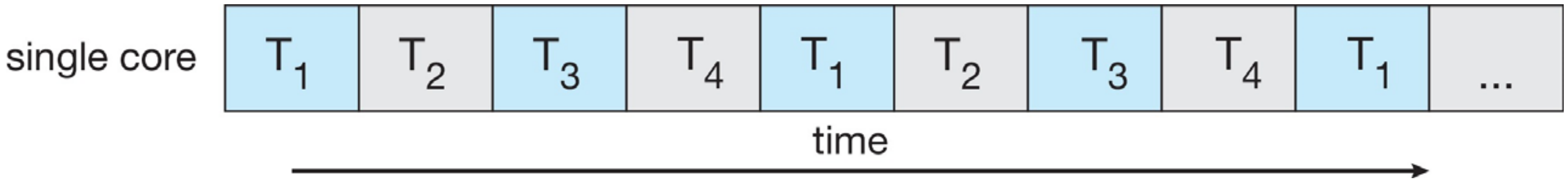
- Responsiveness
  - Allows a program to continue running even if part of it is blocked
    - E.g., a web browser allows user interaction while loading the text/image
- Utilization of MP Architectures
  - Threads can run in parallel on different processors
  - Allows a multithreaded application to run on top of multiple processors
- Resource Sharing
  - Threads share memory & resources
  - Lightweight communication through memory sharing
- Economy
  - More economic to create/switch threads
  - In Solaris, process creation is 32x slower, process switching is 5x slower
- The former two also apply to multi-process architectures, while the latter two are more specific to multi-threading.

Multiprocessor Programming

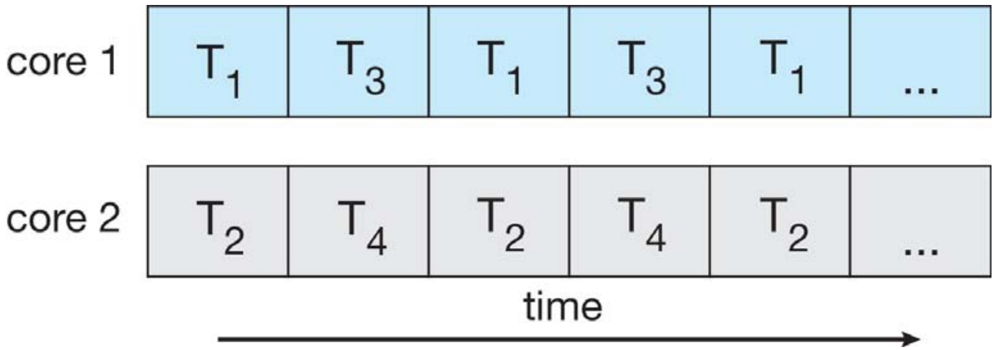
- **Multicore** or **multiprocessor** systems putting pressure on programmers, challenges include:
  - Dividing activities
  - Balance
  - Data splitting
  - Data dependency
  - Testing and debugging
- ***Parallelism*** implies a system can perform more than one task simultaneously
- ***Concurrency*** supports more than one task making progress
  - Single processor/core, scheduler providing concurrency

Concurrency vs. Parallelism

- **Concurrent execution on single-core system:**

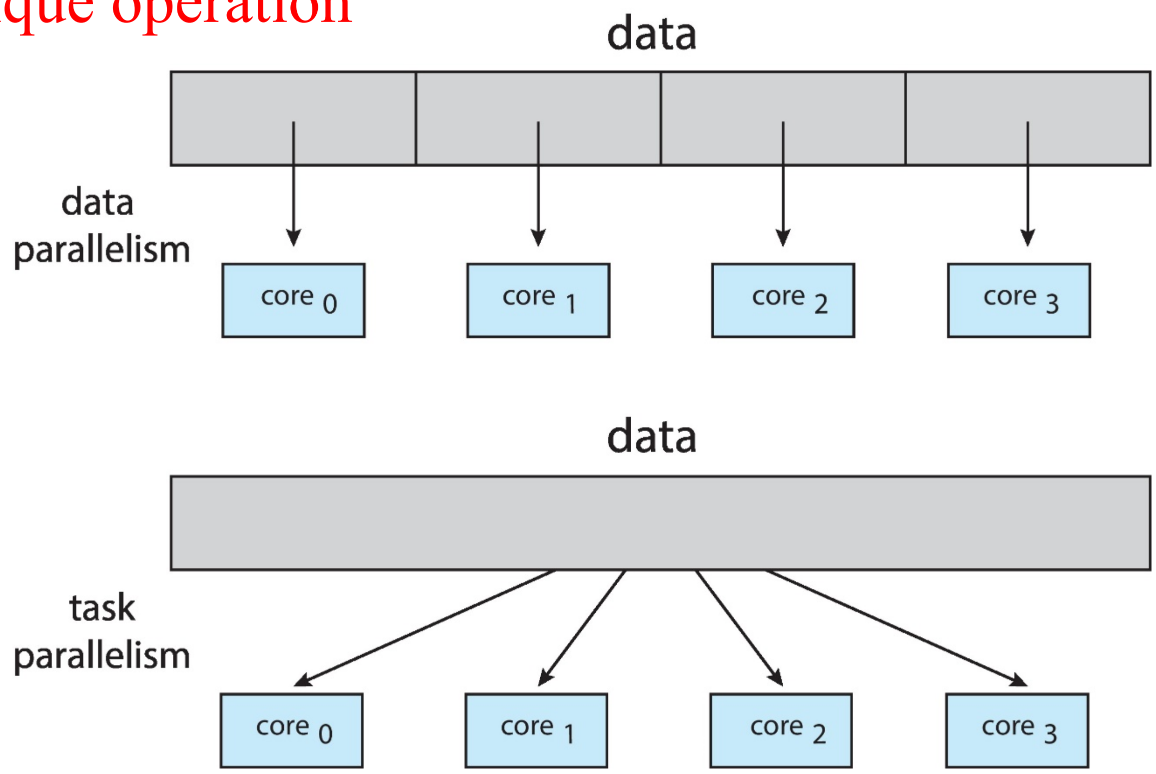


- **Parallelism on a multi-core system:**



Multiprocessor Programming

- Types of parallelism
  - **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
  - **Task parallelism** – distributing threads across cores, each thread performing unique operation



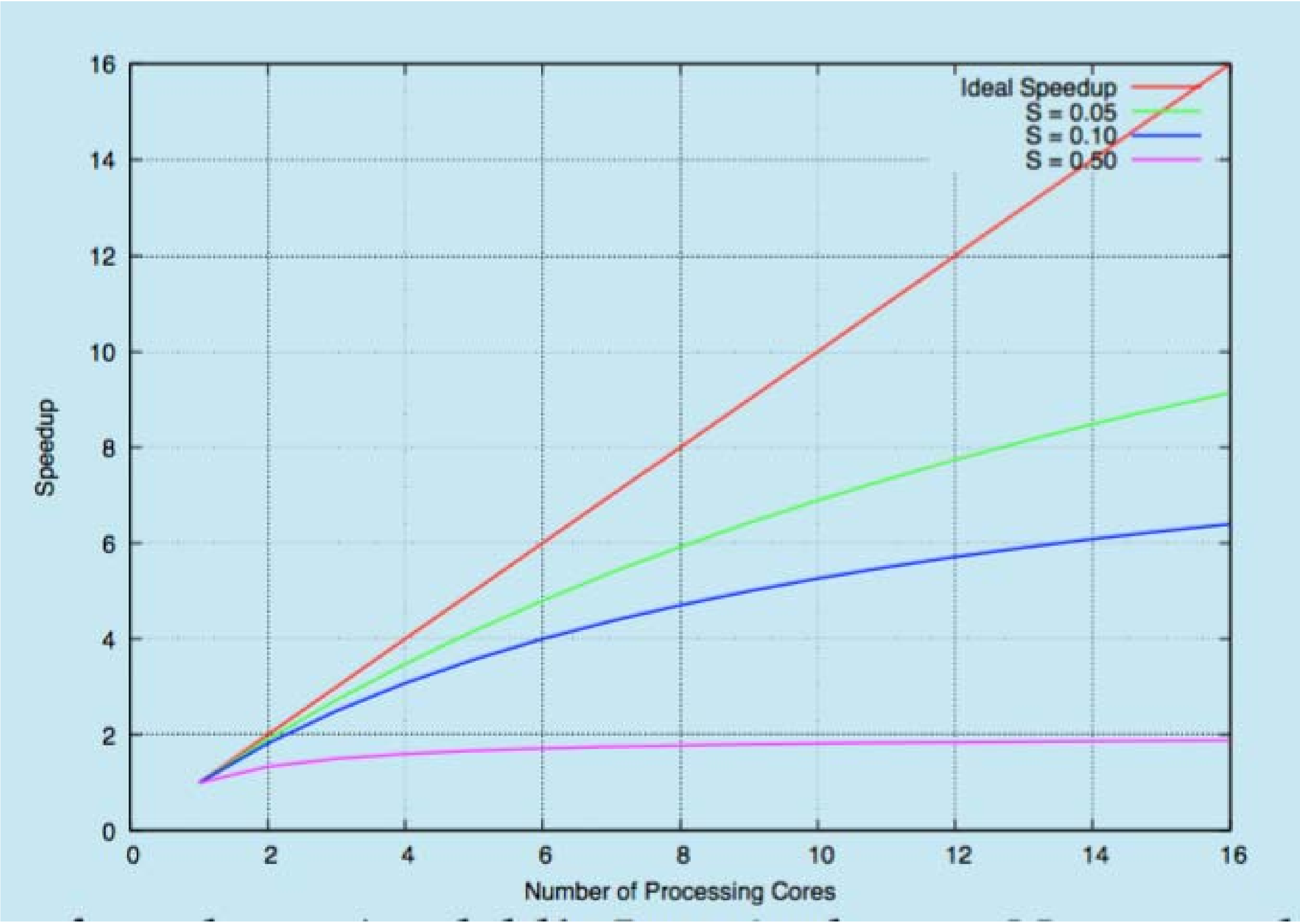
Amdahl's Law

- Identifies performance gains from adding additional cores to an application that has both serial and parallel components
- S is serial portion
- N processing cores

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

- An example
  - if application is 75% parallel and 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times
- As N approaches infinity, speedup approaches 1 / S

Amdahl's Law


Ideal Speedup
S = 0.05
S = 0.10
S = 0.50
Speedup
Number of Processing Cores
0
2
4
6
8
10
12
14
16

Thread Types

- User threads
- Kernel threads

User Threads

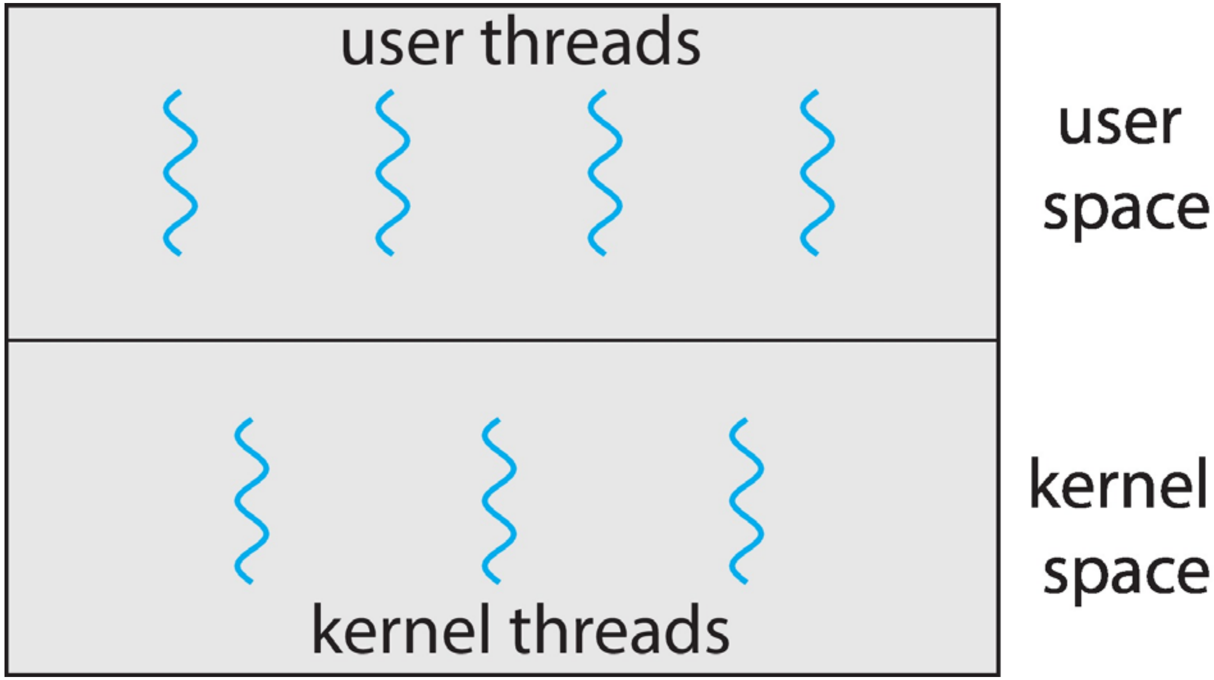
- User-level threads
- Thread management done by **thread library**
- Common thread libraries (***not always user-level implementations, described later***):
  - POSIX Pthreads
    - POSIX: Portable Operating System Interface
  - Windows threads
  - Java threads

Kernel Threads

- Supported by the Kernel

- Examples
  - Windows
  - Solaris
  - Linux
  - Tru64 UNIX (formerly Digital UNIX)
  - Mac OS X

User and Kernel Threads



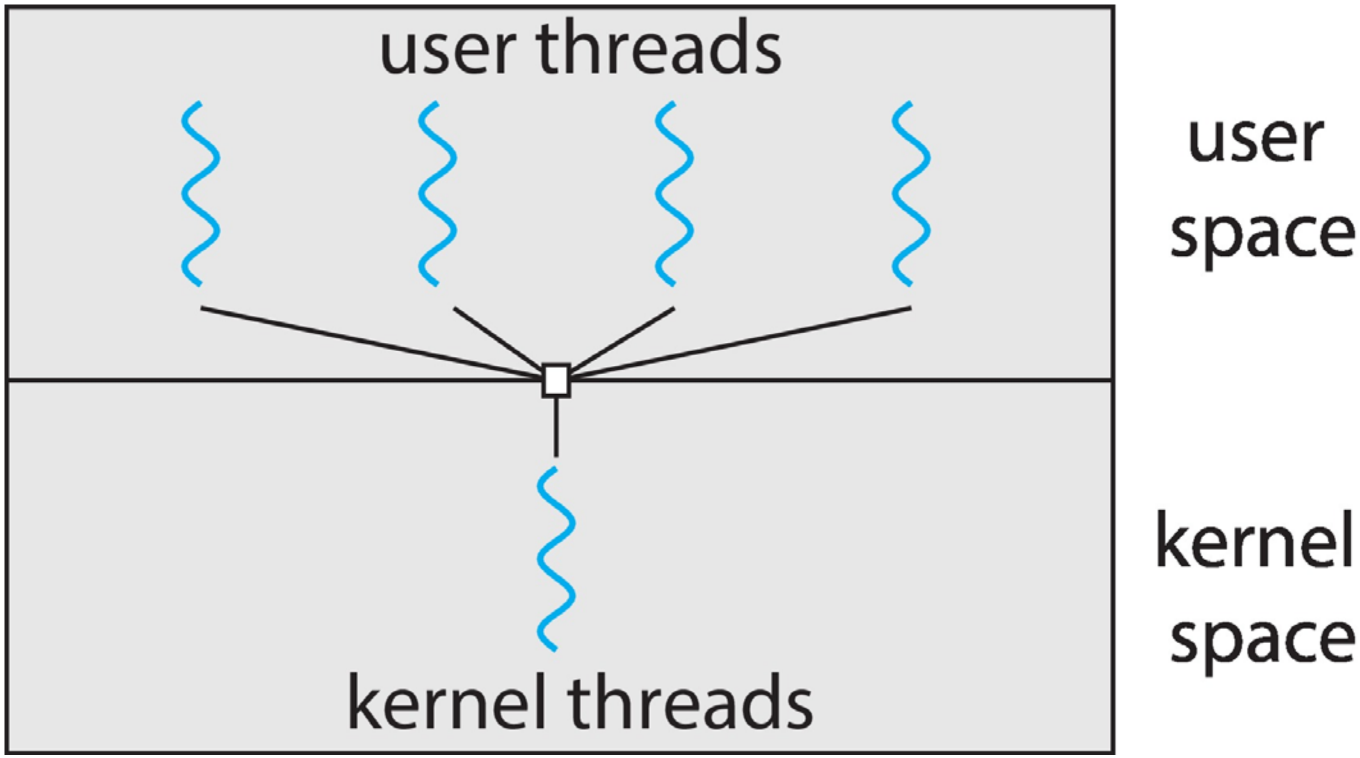
Multithreading Models

- Many-to-One
- One-to-One
- Many-to-Many

Many-to-One

- Many user-level threads mapped to single kernel thread
- Thread management is done by the thread library in the user space
  - Efficient on thread management
  - If one thread makes a blocking system call, the other threads mapping to the same kernel thread will block
  - Multiple threads are unable to run in parallel on multiprocessors
    - since the kernel only sees a single thread (for these user threads)
- Few systems currently use this model
- Examples
  - Solaris Green Threads
  - GNU Portable Threads

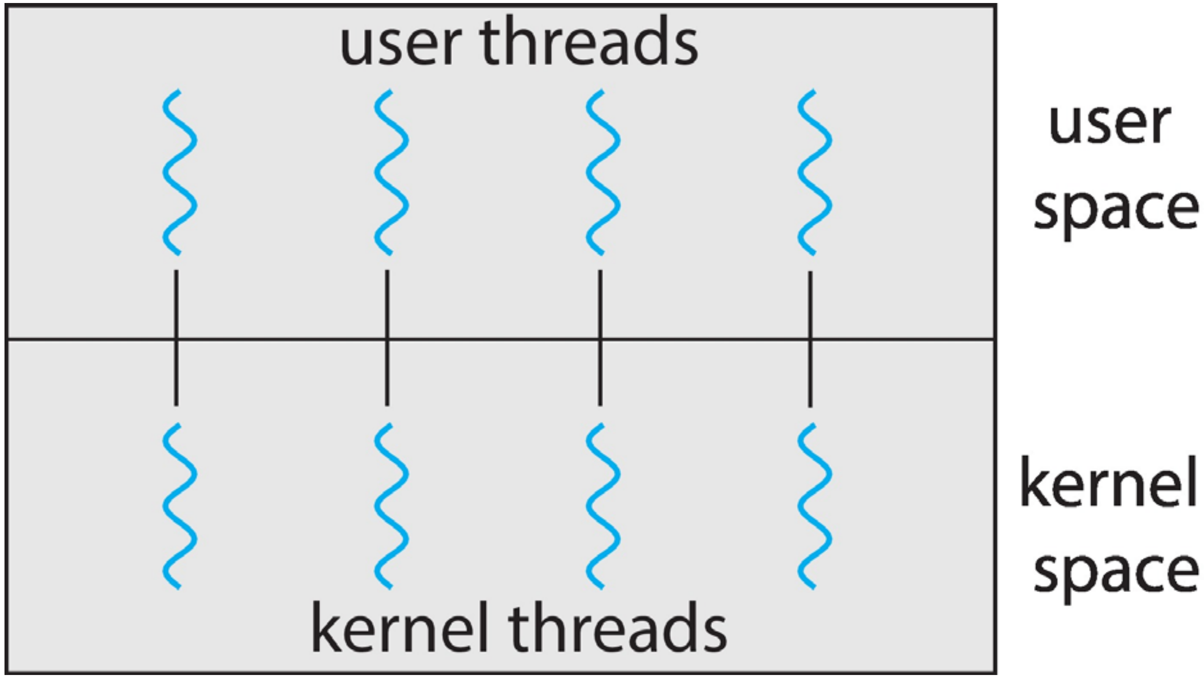
Many-to-One Model



One-to-One

- Each user-level thread maps to a kernel thread
- Allows another thread to run when a thread makes a blocking system call
- Multiple threads can run in parallel on multiprocessors
- The overhead of the kernel threads may burden the performance of the applications
    - Therefore, sometimes you can not create as many threads as necessary
- Examples
  – Windows
  – Linux
  – Solaris 9 and later

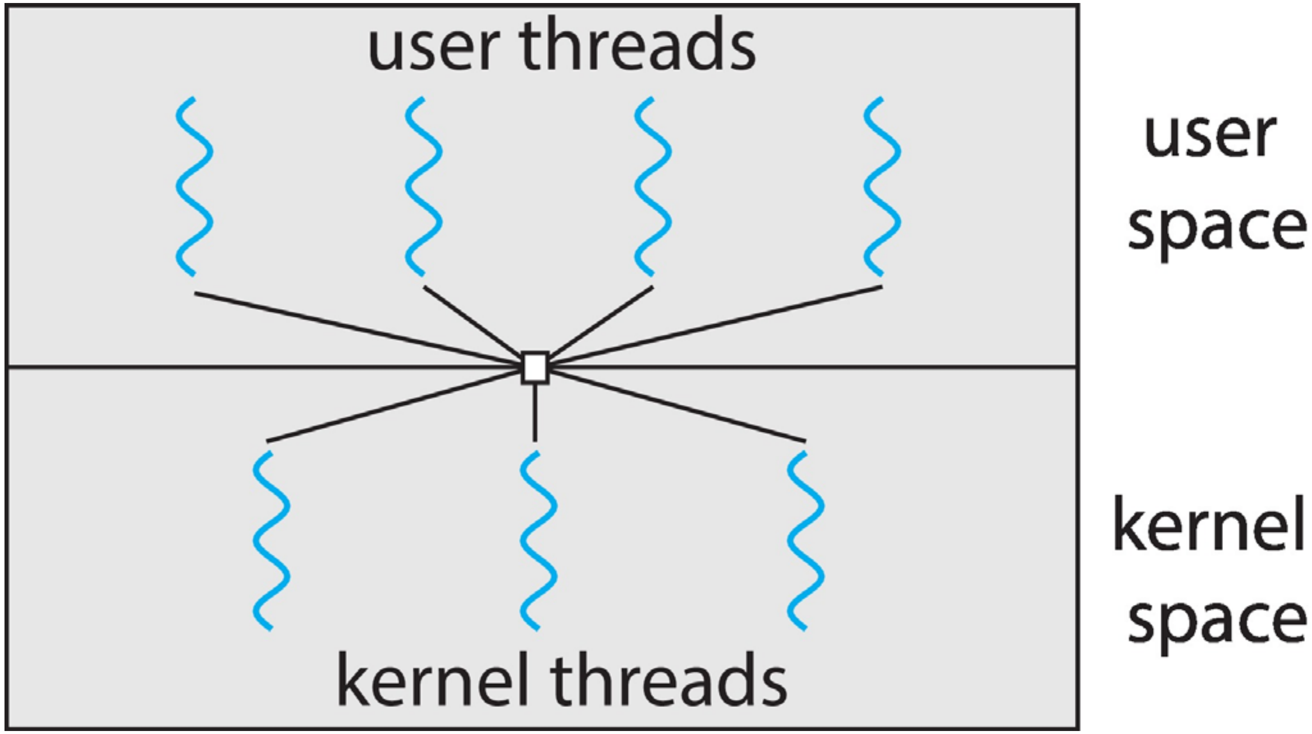
One-to-one Model



Many-to-Many Model

- Allows M user level threads to be mapped to N kernel threads, where N <= M
- Allows the operating system to create a sufficient number of kernel threads
- Allows applications to create as many user threads as necessary
- The kernel threads can run in parallel on multiprocessors
- A blocking system call does not block the entire process
- Examples
  - Solaris prior to version 9
  - Windows NT/2000 with the *ThreadFiber* package

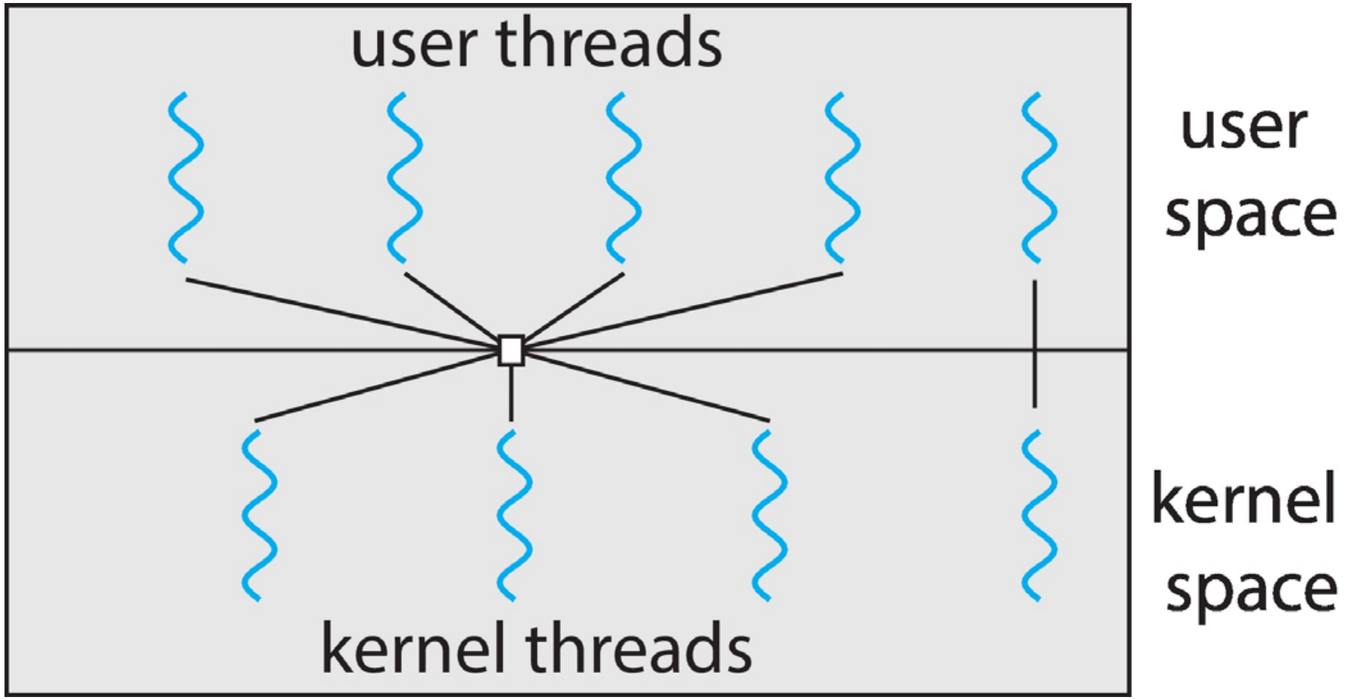
Many-to-Many Model



Two-level Model

- Similar to M:M, except that it allows a user thread to be **bound** to kernel thread
- Examples
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier

Two-level Model



Thread Libraries

- Provides an API for creating and managing threads
- Two primary ways of implementation
  - User-level library
    - Code and data structures reside entirely in the user space
    - A library call does not result in a system call
  - Kernel-level library
    - Code and data structures reside in the kernel space
    - A library call typically results in a system call

Thread Libraries

- Common Thread Libraries
  - Pthreads
    - Provided as either a user- or kernel-level library
  - Windows threads
    - Kernel-level library
  - Java threads
    - Create and manage java threads
    - Use native threading support, typically
      - Use Windows threads in Windows
      - Use Pthreads in UNIX or Linux

Pthreads

- POSIX threads
  - A specification for thread behavior (IEEE 1003.1c)
    - Not an implementation!!!
  - OS designers can implement the specification
  - Numbers of implementations in
    - Solaris, Linux, Mac OS X, Tru64 UNIX
  - Shareware implementations in Windows
- A Pthreads example (see next two slides)
  - A thread begins with main()
  - main() creates a second thread by **pthread_create()**
  - Both threads share the global variable *sum*
  - Wait for a thread to terminate by **pthread_join()**

A Pthreads Example

```
int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* the thread */
main(int argc, char *argv[])
{
   pthread_t tid; /* the thread identifier */
   pthread_attr_t attr; /* set of thread attributes */
   if (argc != 2) {
      fprintf(stderr,"usage:  a.out <integer value>\n");
      exit();
   }
   if (atoi(argv[1]) < 0) {
      fprintf(stderr,"%d must be <= 0\n",atoi(argv[1]));
      exit();
   }
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* create the thread */
   pthread_create(&tid,&attr,runner,argv[1]);
   /* now wait for the thread to exit */
   pthread_join(tid,NULL);
   printf("sum = %d\n",sum);
}
```

A Pthreads Example (cont.)

```
/* The thread will begin control in this function */
void *runner(void *param)
{
   int upper = atoi(param);
   int i;
   sum = 0;
   if (upper > 0) {
      for (i = 1; i <= upper; i++)
         sum += i;
   }
   pthread_exit(0);
}
```

Windows Threads

- Threads are created by CreateThread()
- WaitforSingleObject()
  – Wait for the specified thread to terminate

Windows Threads

```
int main(int argc, char *argv[])
{
  DWORD ThreadId;
  HANDLE ThreadHandle;
  int Param;

  Param = atoi(argv[1]);
  /* create the thread */
  ThreadHandle = CreateThread(
    NULL, /* default security attributes */
    0, /* default stack size */
    Summation, /* thread function */
    &Param, /* parameter to thread function */
    0, /* default creation flags */
    &ThreadId); /* returns the thread identifier */

   /* now wait for the thread to finish */
  WaitForSingleObject(ThreadHandle,INFINITE);

  /* close the thread handle */
  CloseHandle(ThreadHandle);

  printf("sum = %d\n",Sum);
}
```

Java Threads

- Two techniques to create Java threads
  - Create a new class deriving from the Thread class and override its run() method
  - Define a class that implements the Runnable interface

```
Public interface Runnable
{
        Public abstract void run();
}
```

- Threads are actually created when the start() method of the thread object is invoked
  - The start() method
    - Allocates memory and init a new thread in the JVM
    - Calls the run() method
- Invoke join() method of the thread object to wait for the thread to exit

Implicit Threading

- Growing in popularity
  - as numbers of threads increase, program correctness more difficult with explicit threads
- Creation and management of threads done by compilers and run-time libraries rather than programmers
- Examples
  - OpenMP
  - Intel Threading Building Blocks

OpenMP

- Set of compiler directives and an API for C, C++, FORTRAN
- Provides support for parallel programming in shared-memory environments
- Identifies **parallel regions** – blocks of code that can run in parallel

  #pragma omp parallel

  - create as many threads as there are cores

```
#include <omp.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
  /* sequential code */

  #pragma omp parallel
  {
    printf("I am a parallel region.");
  }

  /* sequential code */

  return 0;
}
```

OpenMP

- Run the for loop in parallel

```
#pragma omp parallel for
for (i = 0; i < N; i++) {
   c[i] = a[i] + b[i];
}
```

Threading Issues

- Semantics of **fork()** and **exec()** system calls
- Thread cancellation
- Signal handling
- Thread pools
- Thread specific data

Semantics of fork() and exec()

- Does **fork()** duplicate only the calling thread or all threads?
- Some UNIX systems support two versions of fork()
  - Duplicate all threads
  - Duplicate the caller thread only
- How about exec()?
  - A thread invokes exec() ➔ entire process will be replaced, including all of the threads
- Which version of fork() should be used?
  - Depends on whether the exec() will be called
  - If it will ➔ just duplicate the caller
  - Otherwise, duplicate all the threads

Thread Cancellation

- Thread Cancellation
  - Terminating a thread **before** it has finished
- Examples
  - Searching a database
    - If one thread finds the result, cancel the other threads
  - Web browser
    - If user press the cancel button, cancel the downloading thread
- Two general approaches for thread cancellation
  - **Asynchronous cancellation** terminates the target thread immediately
    - May have problems when
      - Resources have been allocated to the target thread
      - The target thread is updating a shared data
  - **Deferred cancellation** allows the target thread to check later if it should be cancelled
    - Allows a thread to be cancelled in a safe point (**cancellation point**)

Thread Cancellation (Cont.)

- Cancellation mode depends on *state* and *type*

Mode	State	Type
Off	Disabled	–
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

  – If a thread has cancellation state disabled, cancellation remains pending until the thread enables it

- Cancellation points for deferred cancellation
  – certain functions must, and certain other functions may, be cancellation points
    - For example, open()/close()/read()/write() are cancellation points
    - See http://man7.org/linux/man-pages/man7/pthreads.7.html

  – Insert a cancellation point: **pthread_testcancel()**

Signal Handling

- Signals are used in UNIX systems to notify a process that a particular event has occurred
- A signal may be received either synchronously or asynchronously
  - Synchronous signals
    - E.g., illegal memory access, divided by 0
    - Generated by the process itself
  - Asynchronous signals
    - E.g., terminate a process with Ctrl-C, timer expires
    - Generated by an external event

Signal Handling

- A **signal handler** is used to process signals
  - Every signal has a default handler that kernel runs when handling signal
  - User-defined signal handler can override default
- Signal generation & handling
  1. Signal is generated by particular event
  2. Signal is delivered to a process (signal destination)
  3. Signal is handled
- Signal destination for a multi-threaded process
  - Deliver the signal to the thread to which the signal applies
  - Deliver the signal to every thread in the process
  - Deliver the signal to certain threads in the process
  - Assign a specific thread to receive all signals for the process

***A signal can be handled only once!**

Signal Handling

- For syn signals, it should be delivered to the thread that causes the signal
- For asyn signals, it depends…
  - E.g., Ctrl-C ➔ the signal should be sent to all the threads
- Many OSes allows a thread to specify which signals to accept and which signals to block
  - A signal can be delivered to the first found thread that accepts it
    - A signal can be handled only once!

Signal Handling

- Sending a signal to a process in UNIX
  - kill (pid, signal)
- Sending a signal to a thread in Pthreads
  - pthread_kill (tid, signal)
- Asynchronous Procedure Call (APC)
  - in Windows systems
  - Similar to asyn signals
  - Allows a thread to specify a function to be called when an event happens
  - delivered to a thread, not a process

Thread Pools

- Consider a web server that serves each request with a separated thread
  - Thread creation and termination are not free
  - No upper limit on the number of threads
- Solution: Thread Pool
  - Create a number of threads in a pool where they await work
- Advantages
  - Usually slightly faster to serve a request with an existing thread than create a new thread
  - Allows the number of threads in the application(s) to be bound to the size of the pool

Thread Pools

- The number of threads in the pool can be based on
  - Number of CPUs
  - Amount of physical memory
  - Expected number of requests
- Some thread pool architectures can adjust the number of threads in the pool according to usage patterns or load

Thread-Specific Data

- Also called Thread-Local Storage (TLS)

- Allows each thread to have its own copy of data

- Thread specific data is private to a thread, but shared among the functions invoked by the thread
- Most thread libraries support this feature
- Two pthreads functions for thread specific data
  - pthread_setspecific()
  - pthread_getspecific()

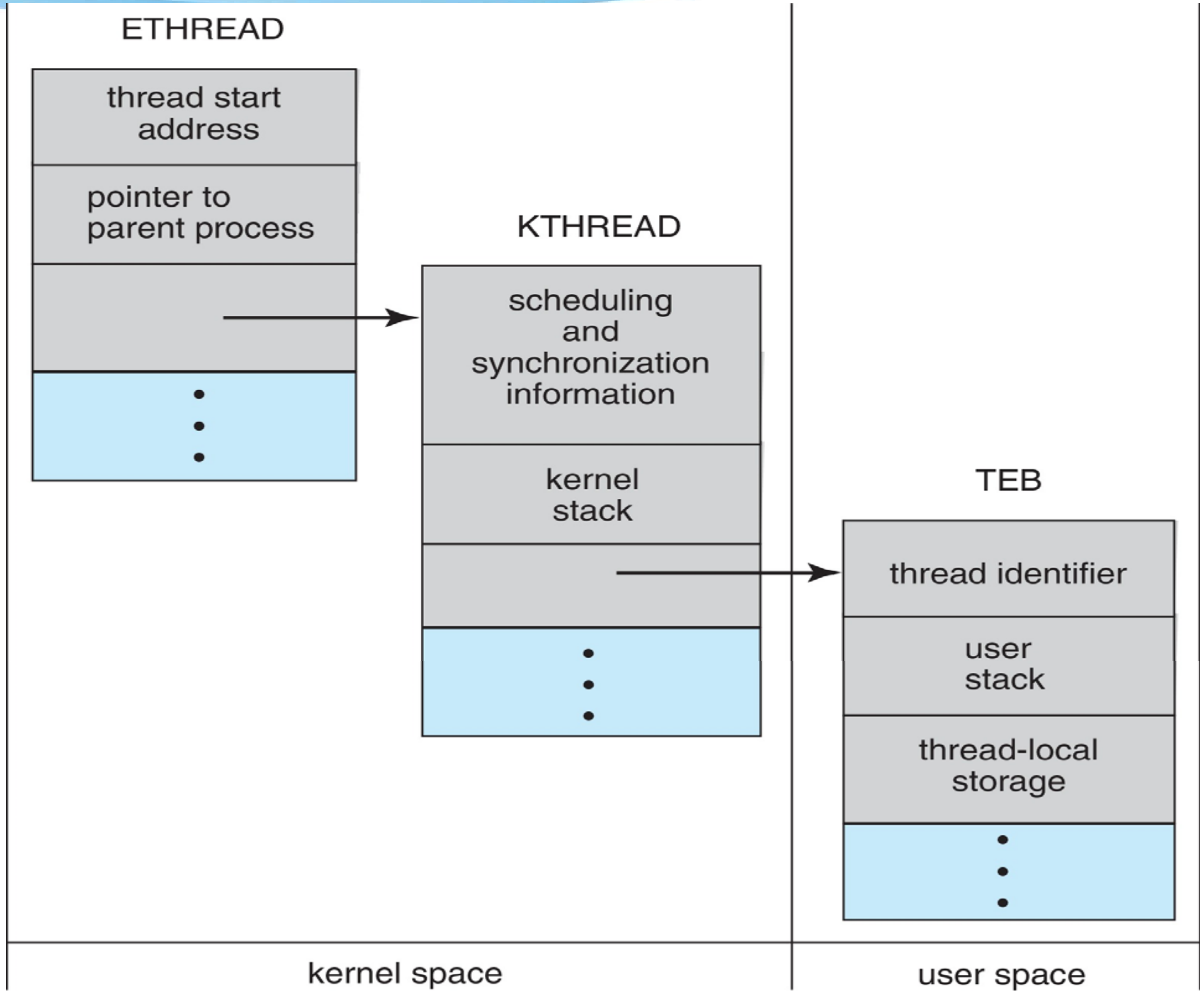
Thread Examples

- Explore how threads are implemented in Windows and Linux systems.

Windows Threads

- Implements the one-to-one mapping
  - also support a fiber library that provides the M:M model
- Each thread contains
  - A thread id
  - Register set
  - Separate user and kernel stacks
  - Private data storage area (i.e. thread specific data)
- The register set, stacks, and private storage area are known as the **context** of the threads
- The primary data structures of a thread include:
  - ETHREAD (executive thread block)
  - KTHREAD (kernel thread block)
  - TEB (thread environment block)

Data Structures of a Windows Thread



Linux Threads

- Linux refers to them as *tasks* rather than *threads* or *processes*
- Thread creation is done through **clone()** system call
- **clone()** allows a child task to share the address space of the parent task (process)

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

Java Threads

- Java threads are managed by the JVM
- Java threads may be created by
  - Extending Thread class
  - Implementing the Runnable interface

Java Thread States